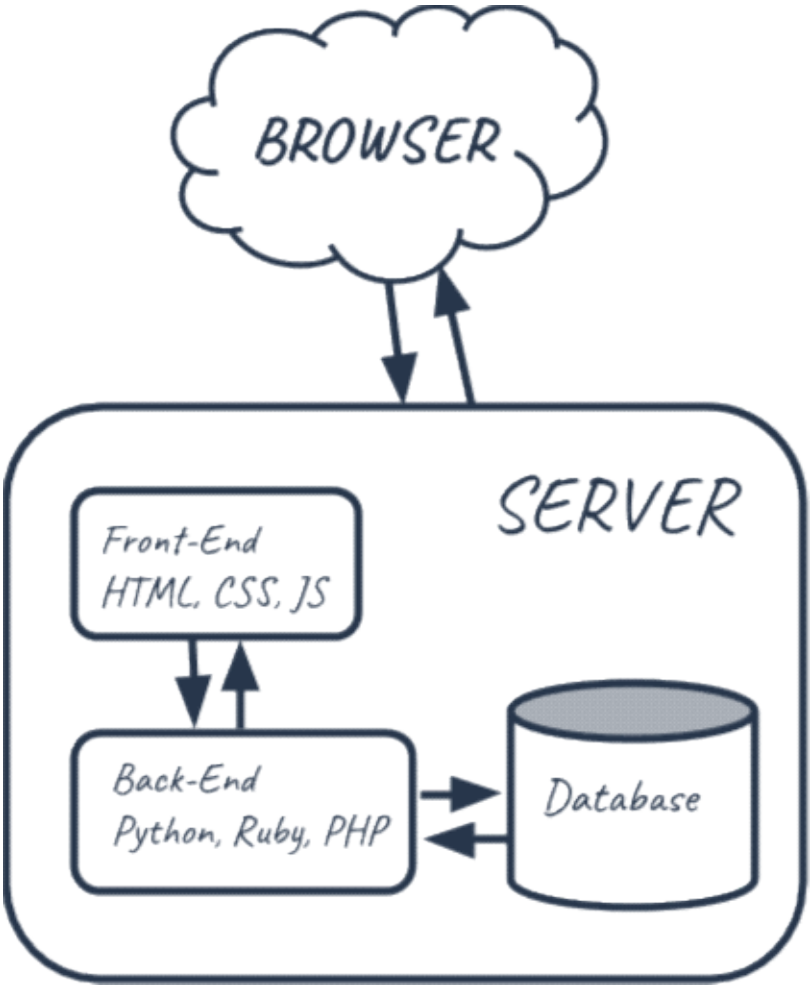
终极以太坊 Dapp 教程（如何逐步构建全栈去中心化应用程序）

格雷戈里·麦克库宾

什么是区块链？

让我们用一个类比来理解区块链是什么以及它是如何工作的。让我们看一个 Web 应用程序。



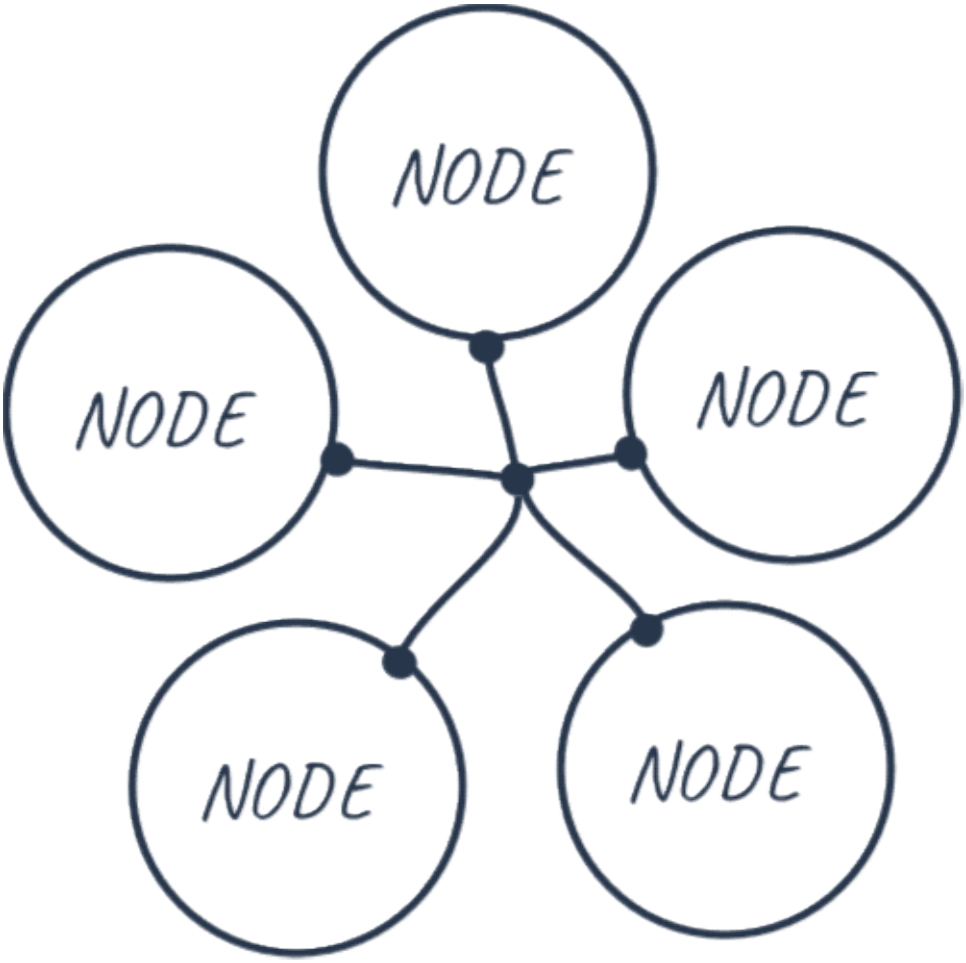
通常，当您与 Web 应用程序交互时，您使用 Web 浏览器通过网络连接到中央服务器。此 Web 应用程序的所有代码都位于此中央服务器上，所有数据位于中央数据库中。无论何时与应用程序进行交易，都必须与 Web 上的此中央服务器进行通信。

如果我们要在网络上构建投票应用程序，我们会遇到一些问题：

1. 数据库上的数据可以更改：可以多次计数，也可以完全删除。
2. Web 服务器上的源代码也可以随时更改。

我们不想在网络上构建我们的应用程序。我们希望将其构建在区块链上，任何连接到网络的人都可以参与选举。我们希望确保他们的选票被计算在内，并且只计算一次。因此，让我们来看看它是如何工作的。

区块链不是网络、中央服务器和数据库，而是集网络和数据库于一身。区块链是计算机的点对点网络，称为节点，共享网络中的所有数据和代码。因此，如果您是连接到区块链的设备，那么您就是网络中的一个节点，并且您与网络中的所有其他计算机节点进行通信。您现在拥有区块链上所有数据和代码的副本。不再有中央服务器。只是一堆在同一网络上相互通信的计算机。



在区块链中的节点之间共享的所有交易数据都包含在称为块的记录包中，而不是集中式数据库，这些记录链接在一起以创建公共分类账。这个公共账本代表区块链中的所有数据。公共账本中的所有数据都通过加密哈希进行保护，并通过共识算法进行验证。网络上的节点参与以确保分布在网络上的所有数据副本相同。这是我们在区块链上构建投票应用程序的一个非常重要的原因，因为我们希望确保我们的投票被计算在内，并且它没有改变。

对于我们应用程序的用户来说，在区块链上投票会是什么样子？好吧，对于初学者来说，用户需要一个带有钱包地址的帐户，其中包含一些以太坊的加密货币以太币。一旦他们连接到网络，他们就会投票并支付少量交易费用，将此交易写入区块链。这笔交易费用称为“gas”。每当投票时，网络上的一些节点，称为矿工，竞相完成这笔交易。完成此交易的矿工将获得我们支付投票的以太币。

回顾一下，当我投票时，我支付汽油价格来投票，当我的投票被记录下来时，网络上的一台计算机会获得我的以太币费用。反过来，我相信我的投票被永远准确地记录下来。

因此，同样重要的是要注意，在区块链上投票会花费以太币，但仅仅看到候选人名单并不能。这是因为从区块链读取数据是免费的，但写入数据却不是免费的。

什么是智能合约？

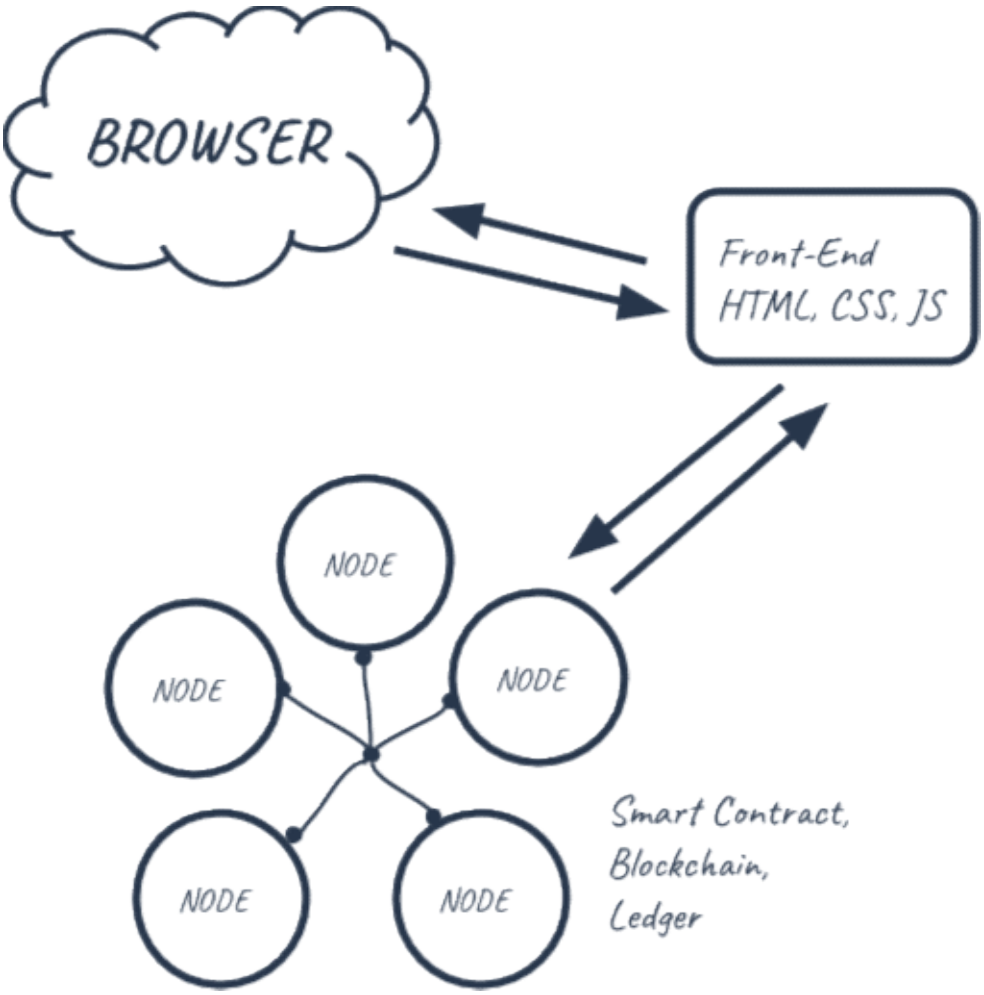
这就是投票过程的工作方式，但是我们如何实际编写应用程序？好吧，以太坊区块链允许我们使用区块链上的以太坊虚拟机（EVM）执行代码，称为智能合约。

智能合约是我们应用程序的所有业务逻辑所在的位置。在这里，我们将实际编写应用程序的去中心化部分。智能合约负责将数据读取和写入区块链，以及执行业务逻辑。智能联系人是用一种名为 Solidity 的编程语言编写的，它看起来很像 Javascript。它是一种成熟的编程语言，它将允许我们做许多与 Javascript 相同类型的事情，但由于它的用例，它的行为略有不同，正如我们将在本教程中看到的那样。

区块链上智能合约的功能与网络上的微服务非常相似。如果公共账本代表区块链的数据库层，那么智能合约就是与该数据进行交易的所有业务逻辑所在的位置。

此外，它们被称为智能合约，因为它们代表契约或协议。就我们的投票 dApp 而言，这是一个协议，我的投票将计算，其他选票只计算一次，并且得票最多的候选人实际上将赢得选举。

现在让我们快速浏览一下我们正在构建的 dApp 的结构。



我们将有一个用 HTML，CSS 和 Javascript 编写的传统前端客户端。该客户端将连接到我们将安装的本地以太坊区块链，而不是与后端服务器通信。我们将使用 Solidity 编程语言在选举智能合约中对有关 dApp 的所有业务逻辑进行编码。我们会将此智能合约部署到我们当地的以太坊区块链，并允许账户开始投票。

现在我们已经看到了区块链是什么以及它是如何工作的。我们已经了解了为什么我们要在区块链而不是当前的网络上构建我们的投票 dApp。我们已经看到，我们希望通过编写将部署到以太坊区块链的智能合约来编写我们的 dApp 代码。现在让我们开始编程吧！

我们将构建什么

这是我们将要构建的投票 dApp 的演示。

我们将构建一个客户端应用程序，该应用程序将与区块链上的智能合约进行通信。此客户端应用程序将有一个候选人表，其中列出了每个候选人的 ID、姓名和投票计数。它将有一个表格，我们可以在其中为我们想要的候选人投票。它还在“您的帐户”下显示我们连接到区块链的帐户。

安装依赖项

本教程这一部分的随附视频片段从 8：53 开始。

为了构建我们的 dApp，我们首先需要一些依赖项。

节点包管理器 （NPM）

我们需要的第一个依赖项是 Node.js 附带的 Node Package Manager（NPM）。您可以通过转到终端并键入以下内容来查看是否已安装节点：

```
$ node -v
```

松露框架

下一个依赖项是 Truffle 框架，它允许我们在以太坊区块链上构建分散的应用程序。它提供了一套工具，使我们能够使用 Solidity 编程语言编写智能联系人。它还使我们能够测试我们的智能合约并将其部署到区块链中。它还为我们提供了一个开发客户端应用程序的地方。

您可以在命令行中安装 Truffle with NPM，如下所示：

```
$ npm install -g truffle
```

甘纳许

下一个依赖项是 Ganache，一个本地内存区块链。您可以通过从 Truffle Framework 网站下载来安装 Ganache。它将为我们提供 10 个外部帐户，其地址位于我们本地以太坊区块链上。每个账户都预装了 100 个假以太币。

小狐狸钱包

下一个依赖项是 Google Chrome 的 Metamask 扩展。为了使用区块链，我们必须连接到它（记住，我说过区块链**是一个**网络）。我们必须安装一个特殊的浏览器扩展才能使用以太坊区块链。这就是小狐狸钱包的用武之地。我们将能够使用我们的个人帐户连接到我们当地的以太坊区块链，并与我们的智能合约进行交互。

我们将在本教程中使用小狐狸钱包 chrome 扩展，因此如果您还没有谷歌浏览器，您还需要安装它。要安装小狐狸钱包，请在谷歌浏览器网上商店中搜索小狐狸钱包 Chrome 插件。安装后，请确保在扩展列表中选中它。安装 Chrome 浏览器后，您会在浏览器的右上角看到狐狸图标。如果您遇到困难，请参考视频演练！

语法突出显示

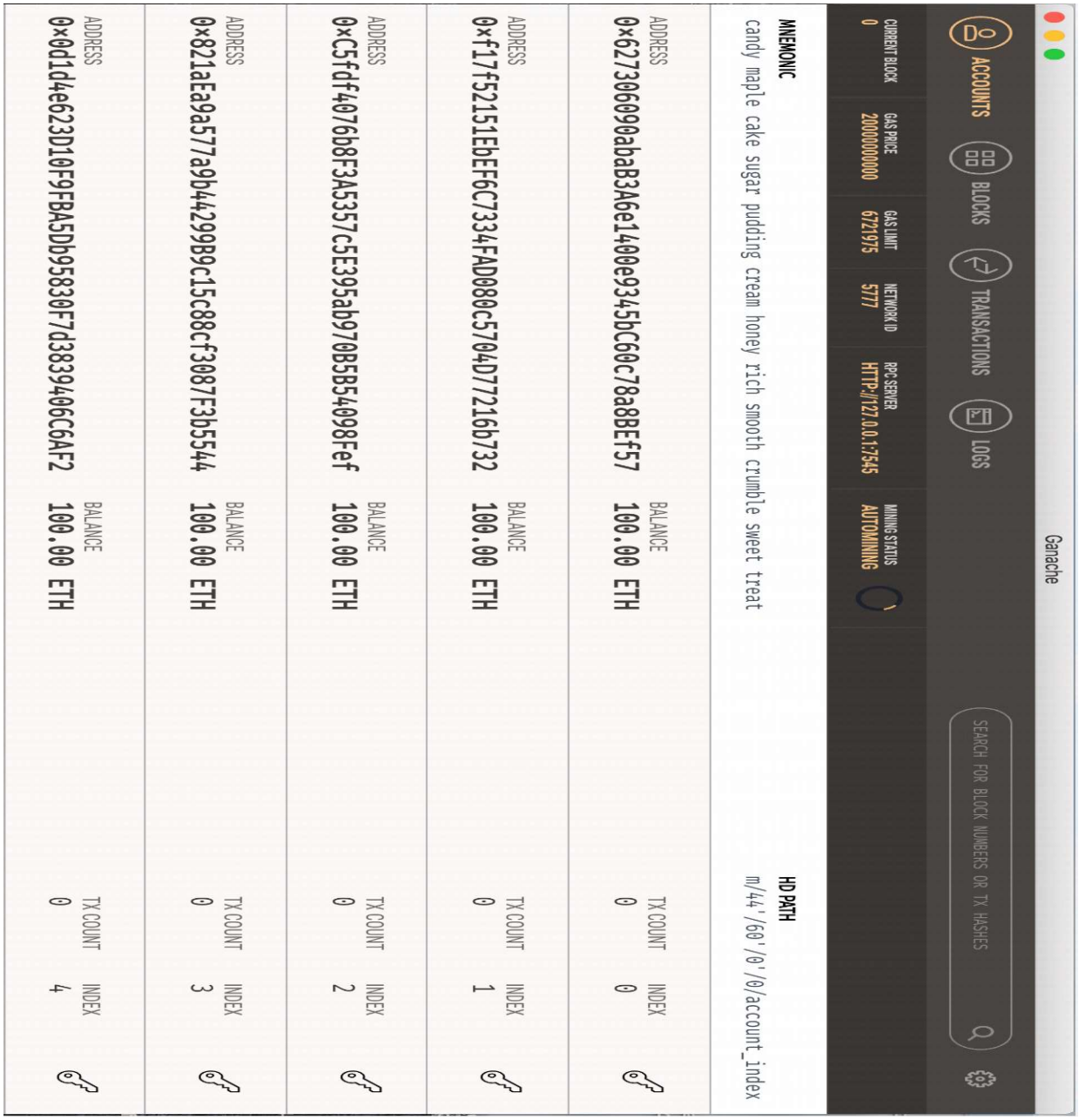
依赖项是可选的，但建议使用。我建议为 Solidity 编程语言安装语法高亮显示。大多数文本编辑器和 IDE 没有开箱即用的 Solidity 语法突出显示功能，因此您必须安装一个软件包来支持此功能。我正在使用 Sublime Text，并且我已经下载了“以太坊”包，该包为 Solidity 提供了很好的语法突出显示。

Smoke Test - Step 1

本教程这一部分的随附视频片段从 11：40 开始。您可以在此处下载本教程此部分的代码。如果您遇到困难，请随时使用这些作为参考点！

现在我们已经安装了依赖项，让我们开始构建我们的 dApp！

首先，找到您下载甘纳许的位置，然后打开它。现在 Ganache 已经启动，您有一个本地区块链正在运行。



Ganache 给了我们 10 个预装了 100 个假以太币的账户（这在以太坊主网络上一文不值）。每个帐户都有一个唯一的地址和一个私钥。每个帐户地址将作为我们选举中每个选民的唯一标识符。

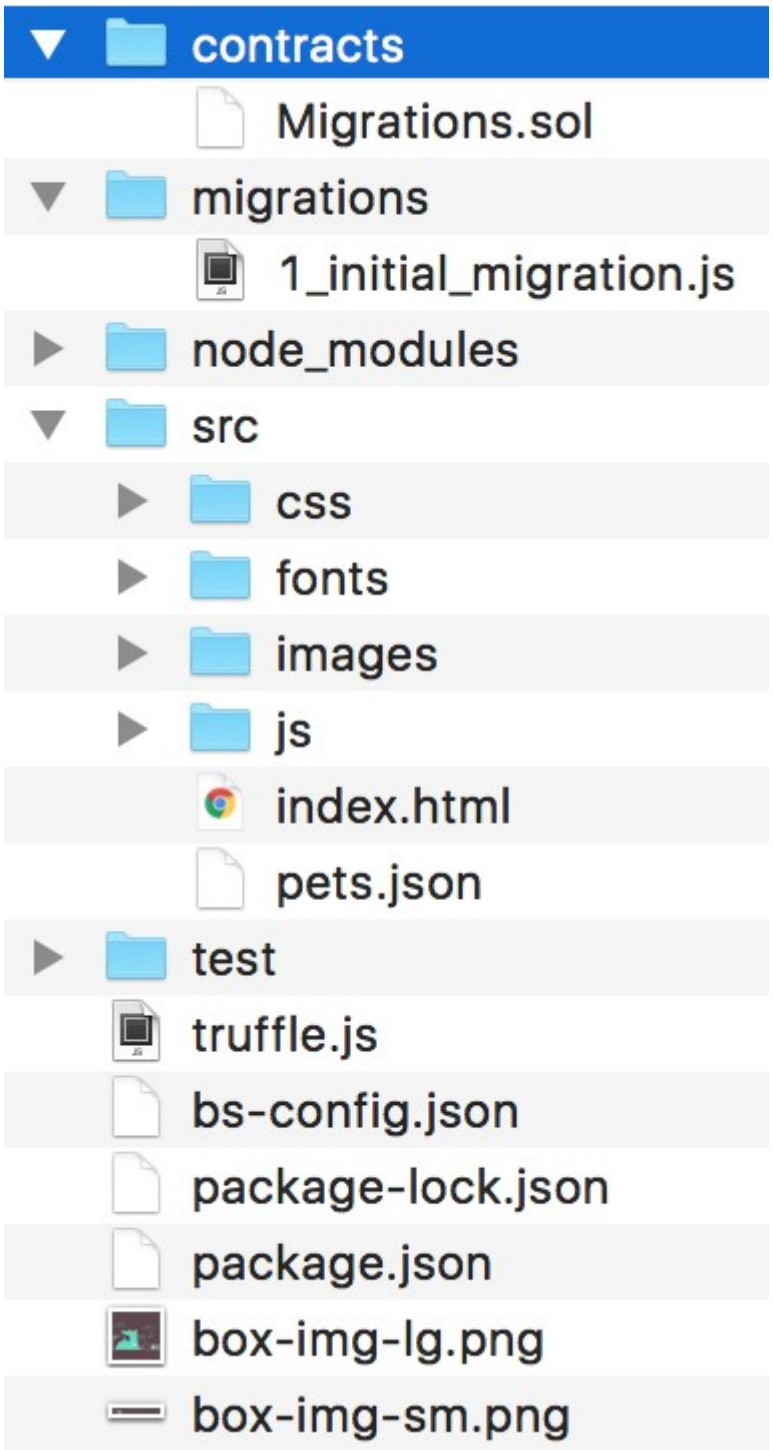
现在让我们在命令行中为我们的 dApp 创建一个项目目录，如下所示：

```
$ mkdir election
$ cd election
```

现在我们已经进入了我们的项目，我们可以使用松露盒快速启动并运行。在本教程中，我们将使用宠物店框。在项目目录中，从命令行安装宠物店盒子，如下所示：

```
$ truffle unbox pet-shop
```

让我们看看宠物店盒子给了我们什么：



- **合同目录：**这是所有智能联系人所在的位置。我们已经有一个迁移合同来处理我们向区块链的迁移。

- 迁移目录：这是所有**迁移**文件所在的位置。这些迁移类似于需要迁移来更改数据库状态的其他 Web 开发框架。每当我们将智能合约部署到区块链时，我们都会更新区块链的状态，因此需要迁移。
- **node_modules 目录：**这是我们所有 Node 依赖项的主页。
- **SRC 目录：**我们将在这里开发客户端应用程序。
- 测试**目录：**这是我们为智能合约编写测试的地方。
- **松露.js 文件：**这是我们松露项目的主要配置文件

现在让我们开始编写智能合约！这个智能合约将包含我们 dApp 的所有业务逻辑。它将负责读取和写入以太坊区块链。它将使我们能够列出将参加选举的候选人，并跟踪所有选票和选民。它还将管理选举的所有规则，例如强制帐户只投票一次。从项目的根目录开始，继续在合约目录中创建一个新的合约文件，如下所示：

```
$ touch contracts/Election.sol
```

让我们从创建一个“冒烟测试”开始，以确保我们正确设置了项目，并且我们可以成功地将合约部署到区块链。打开文件并从以下代码开始：

```
pragma solidity 0.4.2;

contract Election {
    // Read/write candidate
    string public candidate;

    // Constructor
    function Election () public {
        candidate = "Candidate 1";
    }
}
```

让我解释一下这段代码。我们首先用语句声明 solidity 版本。接下来，我们使用“contract”关键字声明智能合约，后跟合约名称。接下来，我们声明一个状态变量，该变量将存储候选名称的值。状态变量允许我们将数据写入区块链。我们已经声明了这个变量将是一个字符串，并且我们将其可见性设置为 .因为它是公开的，solidity 将免费为我们提供一个 getter 函数，允许我们在合同之外访问此值。稍后我们将在控制台中看到它的实际效果！`pragma solidity`public

然后，我们创建一个构造函数，每当我们将智能合约部署到区块链时，都会调用该函数。在这里，我们将设置候选状态变量的值，该变量将在迁移时存储到区块链中。请注意，构造函数与智能合约同名。这就是 Solidity 知道函数是构造函数的方式。

现在我们已经为智能合约创建了基础，让我们看看是否可以将其部署到区块链中。为此，我们需要在迁移目录中创建一个新文件。从项目根目录，从命令行创建一个新文件，如下所示：

```
$ touch migrations/2_deploy_contracts.js
```

请注意，我们用数字对迁移目录中的所有文件进行编号，以便 Truffle 知道以哪个顺序执行它们。让我们创建一个新的迁移来部署合约，如下所示：

```
var Election = artifacts.require("./Election.sol");
module.exports = function(deployer) {
  deployer.deploy(Election);
};
```

首先，我们需要我们创建的合约，并将其分配给一个名为“Election”的变量。接下来，我们将它添加到已部署协定的清单中，以确保在运行迁移时部署它。现在让我们像这样从命令行运行迁移：

```
$ truffle migrate
```

现在我们已经成功地将智能合约迁移到本地以太坊区块链，让我们打开控制台与智能合约进行交互。您可以从命令行打开松露控制台，如下所示：

```
$ truffle console
```

现在我们已经进入了控制台，让我们获取已部署智能合约的实例，看看是否可以从合约中读取候选者的名称。在控制台中，运行以下代码：

```
Election.deployed().then(function(instance) { app = instance })
```

这是我们在迁移文件中创建的变量的名称。我们检索了带有函数的合约的已部署实例，并将其分配给 promise 回调函数中的一个变量。乍一看可能有点令人困惑，但您可以参考 21：50 视频中的控制台演示以获取进一步说明。

```
Electiondeployed()app
```

现在我们可以像这样读取候选变量的值：

```
app.candidate()
// => 'Candidate 1'
```

祝贺！您刚刚编写了第一个智能合约，部署到区块链中，并检索了其中的一些数据。

List Candidates - Step 2

本教程这一部分的随附视频片段从 27：11 开始。您可以在此处下载本教程此部分的代码。如果您遇到困难，请随时使用这些作为参考点！

现在一切都设置正确，让我们通过列出将在选举中竞选的候选人来继续构建智能联系人。我们需要一种方法来存储多个候选人，并存储有关每个候选人的多个属性。我们希望跟踪候选人的 ID、姓名和票数。以下是我们将如何对候选人进行建模：

```
contract Election {
    // Model a Candidate
    struct Candidate {
        uint id;
```

```
        string name;
        uint voteCount;
    }

    // ...
}
```

我们已经使用坚固性结构对候选人进行了建模。Solidity 允许我们创建自己的结构类型，就像我们在这里为候选人所做的那样。我们指定此结构的 id 为无符号整数类型、字符串类型的名称和无符号整数类型的 voteCount。简单地声明这个结构实际上不会给我们一个候选者。我们需要实例化它并将其分配给一个变量，然后才能将其写入存储。

接下来我们需要的是一个存储候选人的地方。我们需要一个地方来存储我们刚刚创建的结构类型之一。我们可以使用 Solidity 映射来做到这一点。Solidity 中的映射类似于关联数组或哈希，用于关联键值对。我们可以像这样创建这个映射：

```
contract Election {
    // Model a Candidate
    struct Candidate {
        uint id;
        string name;
        uint voteCount;
    }

    // Read/write Candidates
    mapping(uint => Candidate) public candidates;

    // ...
}
```

在本例中，映射的键是一个无符号整数，该值是我们刚刚定义的 Candidate 结构类型。这基本上为我们提供了每个候选人的基于 ID 的查找。由于此映射被分配给状态变量，因此每当我们为其分配新的键值对时，我们都会将数据写入区块链。接下来，我们将此映射的可见性设置为获取 getter 函数，就像我们在烟雾测试中对候选名称所做的那样。`public`

接下来，我们使用如下计数器缓存状态变量跟踪选举中存在多少候选人：

```
contract Election {
    // Model a Candidate
    struct Candidate {
        uint id;
        string name;
        uint voteCount;
    }
```

```
    // Read/write Candidates
    mapping(uint => Candidate) public candidates;

    // Store Candidates Count
    uint public candidatesCount;

    // ...
}
```

在 Solidity 中，无法确定映射的大小，也无法迭代映射。这是因为映射中尚未分配值的任何键都将返回默认值（在本例中为空候选项）。例如，如果我们在这次选举中只有 2 名候选人，并且我们尝试查找候选人 #99，则映射将返回一个空的候选人结构。这种行为使得无法知道存在多少候选者，因此我们必须使用计数器缓存。

接下来，让我们创建一个函数，将候选项添加到我们创建的映射中，如下所示：

```
contract Election {
    // ...

    function addCandidate (string _name) private {
        candidatesCount ++;
        candidates[candidatesCount] = Candidate(candidatesCount, _name,
0);
    }
}
```

我们已经声明了一个函数，该函数采用一个表示候选项名称的字符串类型的参数。在函数内部，我们递增候选计数器缓存以表示已添加新候选。然后，我们使用新的候选结构更新映射，使用当前候选计数作为键。此候选结构使用当前候选计数中的候选 id、函数参数中的名称以及初始投票计数为 0 进行初始化。请注意，此函数的可见性是私有的，因为我们只想在合约中调用它。addCandidate

现在，我们可以通过在构造函数中调用两次“addCandidate”函数来将两个候选人添加到我们的选举中，如下所示：

```
contract Election {
    // ...

    function Election () public {
        addCandidate("Candidate 1");
        addCandidate("Candidate 2");
    }

    // ...
}
```

当我们将合约部署到区块链时，此迁移将执行，并用两个候选人填充我们的选举。此时，您的完整合约代码应如下所示：

```
pragma solidity ^0.4.2;

contract Election {
    // Model a Candidate
    struct Candidate {
        uint id;
        string name;
        uint voteCount;
    }

    // Read/write candidates
    mapping(uint => Candidate) public candidates;

    // Store Candidates Count
    uint public candidatesCount;

    function Election () public {
        addCandidate("Candidate 1");
        addCandidate("Candidate 2");
    }

    function addCandidate (string _name) private {
        candidatesCount ++;
        candidates[candidatesCount] = Candidate(candidatesCount, _name,
0);
    }

}
```

现在让我们像这样迁移我们的合约：

```
$ truffle migrate --reset
```

现在尝试在控制台内与候选人进行交互。你可以和我一起关注我在 37：31 的视频中演示这一点。我把它留给你作为练习。

现在让我们编写一些测试来确保我们的智能合约正确初始化。首先，让我解释一下为什么在开发智能合约时测试如此重要。我们希望确保合同没有错误，原因如下：

1. 以太坊区块链上的所有代码都是不可变的;它无法改变。如果合约包含任何错误，我们必须禁用它并部署一个新副本。此新副本将不具有与旧合同相同的状态，并且它将具有不同的地址。

2. 部署合约需要花费 gas，因为它会创建交易并将数据写入区块链。这需要以太币的成本，我们希望尽量减少我们必须支付的以太币数量。

3. 如果我们任何写入区块链的合约函数包含错误，调用此函数的账户可能会浪费以太币，并且它的行为方式可能不符合他们的预期。

测试

现在让我们编写一些测试。确保先运行甘纳许。然后，从项目的根目录在命令行中创建一个新的测试文件，如下所示：

```
$ touch test/election.js
```

我们将用 Javascript 编写所有测试，使用 Mocha 测试框架和 Chai 断言库。这些与 Truffle 框架捆绑在一起。我们将用 Javascript 编写所有这些测试，以模拟客户端与我们的智能合约的交互，就像我们在控制台中所做的那样。以下是测试的所有代码：

```
var Election = artifacts.require("./Election.sol");

contract("Election", function(accounts) {
  var electionInstance;

  it("initializes with two candidates", function() {
    return Election.deployed().then(function(instance) {
      return instance.candidatesCount();
    }).then(function(count) {
      assert.equal(count, 2);
    });
  });

  it("it initializes the candidates with the correct values", function()
{
    return Election.deployed().then(function(instance) {
      electionInstance = instance;
      return electionInstance.candidates(1);
    }).then(function(candidate) {
      assert.equal(candidate[0], 1, "contains the correct id");
      assert.equal(candidate[1], "Candidate 1", "contains the correct
name");
      assert.equal(candidate[2], 0, "contains the correct votes count");
      return electionInstance.candidates(2);
    }).then(function(candidate) {
      assert.equal(candidate[0], 2, "contains the correct id");
      assert.equal(candidate[1], "Candidate 2", "contains the correct
name");
      assert.equal(candidate[2], 0, "contains the correct votes count");
    });
```

```
  });
});
```

让我解释一下这段代码。首先，我们需要合约并将其分配给变量，就像我们在迁移文件中所做的那样。接下来，我们调用“contract”函数，并在回调函数中编写所有测试。这个回调函数提供了一个“accounts”变量，代表我们区块链上的所有帐户，由 Ganache 提供。

第一个测试通过检查候选人计数是否等于 2 来检查合同是否使用正确的候选人数初始化。

下一个测试检查选举中每个候选人的值，确保每个候选人具有正确的 ID、姓名和票数。

现在让我们像这样从命令行运行测试：

```
$ truffle test
```

耶，他们通过了！跟随我，因为我在视频中编写这些测试以进行进一步解释。

客户端应用程序

现在让我们开始构建将与我们的智能合约通信的客户端应用程序。我们将通过修改我们在上一节中安装的松露宠物店盒子附带的 HTML 和 Javascript 文件来实现这一点。我们将使用此现有代码开始。让我们也注意松露宠物店盒子附带的其他一些东西，比如 Bootstrap 框架，这将使我们不必在本教程中编写任何 CSS。我们还获得了精简服务器，它将为我们的资产提供开发目的。

您不必是前端专家即可遵循本教程的这一部分。我有意保持 HTML 和 Javascript 代码非常简单，我们不会花太多时间关注它。我想继续专注于开发我们 dApp 的智能合约部分！

继续并将“index.html”文件的所有内容替换为以下代码：

```
<!DOCTYPE html><html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Election Results</title>

    <!-- Bootstrap -->
    <link href="css/bootstrap.min.css" rel="stylesheet">
  </head>
  <body>
    <div class="container" style="width: 650px;">
      <div class="row">
        <div class="col-lg-12">
```

```
          <h1 class="text-center">Election Results</h1>
          <hr/>
          <br/>
          <div id="loader">
            <p class="text-center">Loading...</p>
          </div>
          <div id="content" style="display: none;">
            <table class="table">
              <thead>
                <tr>
                  <th scope="col">#</th>
                  <th scope="col">Name</th>
                  <th scope="col">Votes</th>
                </tr>
              </thead>
              <tbody id="candidatesResults">
              </tbody>
            </table>
            <hr/>
            <p id="accountAddress" class="text-center"></p>
          </div>
        </div>
      </div>
    </div>

    <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.js"
></script>
    <!-- Include all compiled plugins (below), or include individual files
as needed -->
    <script src="js/bootstrap.min.js"></script>
    <script src="js/web3.min.js"></script>
    <script src="js/truffle-contract.js"></script>
    <script src="js/app.js"></script>
  </body></html>
```

接下来，将“app.js”文件的所有内容替换为以下代码：

```
App = {
  web3Provider: null,
  contracts: {},
  account: '0x0',

  init: function() {
    return App.initWeb3();
```

```
  },

  initWeb3: function() {
    if (typeof web3 !== 'undefined') {
      // If a web3 instance is already provided by Meta Mask.
      App.web3Provider = web3.currentProvider;
      web3 = new Web3(web3.currentProvider);
    } else {
      // Specify default instance if no web3 instance provided
      App.web3Provider = new
Web3.providers.HttpProvider('http://localhost:7545');
      web3 = new Web3(App.web3Provider);
    }
    return App.initContract();
  },

  initContract: function() {
    $.getJSON("Election.json", function(election) {
      // Instantiate a new truffle contract from the artifact
      App.contracts.Election = TruffleContract(election);
      // Connect provider to interact with contract
      App.contracts.Election.setProvider(App.web3Provider);

      return App.render();
    });
  },

  render: function() {
    var electionInstance;
    var loader = $("#loader");
    var content = $("#content");

    loader.show();
    content.hide();

    // Load account data
    web3.eth.getCoinbase(function(err, account) {
      if (err === null) {
        App.account = account;
        $("#accountAddress").html("Your Account: " + account);
      }
    });

    // Load contract data
```

```
    App.contracts.Election.deployed().then(function(instance) {
      electionInstance = instance;
      return electionInstance.candidatesCount();
    }).then(function(candidatesCount) {
      var candidatesResults = $("#candidatesResults");
      candidatesResults.empty();

      for (var i = 1; i <= candidatesCount; i++) {
        electionInstance.candidates(i).then(function(candidate) {
          var id = candidate[0];
          var name = candidate[1];
          var voteCount = candidate[2];

          // Render candidate Result
          var candidateTemplate = "<tr><th>" + id + "</th><td>" + name +
"</td><td>" + voteCount + "</td></tr>"
          candidatesResults.append(candidateTemplate);
        });
      }

      loader.hide();
      content.show();
    }).catch(function(error) {
      console.warn(error);
    });
  }
};

$(function() {
  $(window).load(function() {
    App.init();
  });
});
```

让我们记下这段代码所做的一些事情：

1. 设置 web3：**web3**.js 是一个 JavaScript 库，允许我们的客户端应用程序与区块链通信。我们在“initWeb3”函数中配置 web3。
2. **初始化合约：**我们在此函数中获取智能合约的已部署实例，并分配一些值，以允许我们与之交互。
3. **渲染功能：**渲染函数使用来自智能合约的数据布置页面上的所有内容。现在，我们列出了我们在智能合约中创建的候选人。我们通过循环遍历映射中的每个候选项并将其呈现到表中来做到这一点。我们还获取此函数中连接到区块链的当前帐户并将其显示在页面上。

您可以在 57：21 的视频中观看我更深入地解释此代码。

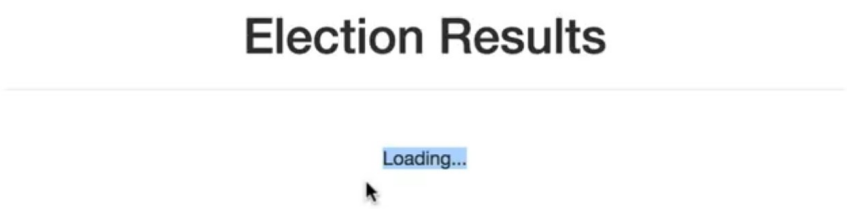
现在让我们在浏览器中查看客户端应用程序。首先，请确保已按如下所示迁移合同：

```
$ truffle migrate --reset
```

接下来，从命令行启动开发服务器，如下所示：

```
$ npm run dev
```

这应该会自动打开一个新的浏览器窗口，其中包含您的客户端应用程序。



请注意，您的应用程序显示“正在加载...”。那是因为我们还没有登录区块链！为了连接到区块链，我们需要将其中一个帐户从甘纳许导入小狐狸钱包。您可以在 1：09：05 的视频中观看我设置 Metamask。

与小狐狸钱包连接后，您应该会看到所有合约和帐户数据已加载。

Election Results

#	Name	Votes
1	Candidate 1	0
2	Candidate 2	0

Your Account: 0x0d80d9ce33320b0f1a0ec0bd75723a06a6fa9d28

投票 - 步骤 3

本教程这一部分的随附视频片段从 1：13：39 开始。您可以在此处下载本教程此部分的代码。如果您遇到困难，请随时使用这些作为参考点！

现在让我们添加在选举中投票的功能。让我们定义一个映射到智能合约的“选民”，以跟踪在选举中投票的账户，如下所示：

```
contract Election {
    // ...

    // Store accounts that have voted
    mapping(address => bool) public voters;

    // ...
}
```

现在让我们添加一个“投票”函数：

```
contract Election {
    // ...

    // Store accounts that have voted
    mapping(address => bool) public voters;

    // ...

    function vote (uint _candidateId) public {
        // require that they haven't voted before
        require(!voters[msg.sender]);

        // require a valid candidate
        require(_candidateId > 0 && _candidateId <= candidatesCount);

        // record that voter has voted
        voters[msg.sender] = true;

        // update candidate vote Count
        candidates[_candidateId].voteCount ++;
    }
}
```

此函数的核心功能是通过从“候选人”映射中读取候选人结构并使用增量运算符（++） 将“voteCount”增加 1 来增加候选人的投票计数。让我们看一下它所做的其他一些事情：

1. 它接受一个参数。这是一个带有候选人 id 的无符号整数。

2. 它的可见性是公开的，因为我们希望外部帐户调用它。
3. 它将投票的帐户添加到我们刚刚创建的选民映射中。这将使我们能够跟踪选民在选举中投票的情况。我们使用 Solidity 提供的全局变量“msg.sender”访问调用此函数的帐户。
4. 它实现了 require 语句，如果不满足条件，这些语句将停止执行。首先要求选民以前没有投票过。我们通过从映射中读取带有“msg.sender”的帐户地址来执行此操作。如果存在，则表示该帐户已投票。接下来，它要求候选 ID 有效。候选 ID 必须大于零且小于或等于候选总数。

现在，您的完整合约代码应如下所示：

```
pragma solidity ^0.4.2;

contract Election {
    // Model a Candidate
    struct Candidate {
        uint id;
        string name;
        uint voteCount;
    }

    // Store accounts that have voted
    mapping(address => bool) public voters;
    // Read/write candidates
    mapping(uint => Candidate) public candidates;
    // Store Candidates Count
    uint public candidatesCount;

    function Election () public {
        addCandidate("Candidate 1");
        addCandidate("Candidate 2");
    }

    function addCandidate (string _name) private {
        candidatesCount ++;
        candidates[candidatesCount] = Candidate(candidatesCount, _name,
0);
    }

    function vote (uint _candidateId) public {
        // require that they haven't voted before
        require(!voters[msg.sender]);

        // require a valid candidate
        require(_candidateId > 0 && _candidateId <= candidatesCount);
```

```
        // record that voter has voted
        voters[msg.sender] = true;

        // update candidate vote Count
        candidates[_candidateId].voteCount ++;
    }
}
```

观看我在 1：13：58 深入解释投票。您还可以观看我在控制台中演示 1：20：38 的投票。

测试投票功能

现在让我们在“election.js”测试文件中添加一个测试：

```
it("allows a voter to cast a vote", function() {
    return Election.deployed().then(function(instance) {
      electionInstance = instance;
      candidateId = 1;
      return electionInstance.vote(candidateId, { from: accounts[0] });
    }).then(function(receipt) {
      return electionInstance.voters(accounts[0]);
    }).then(function(voted) {
      assert(voted, "the voter was marked as voted");
      return electionInstance.candidates(candidateId);
    }).then(function(candidate) {
      var voteCount = candidate[2];
      assert.equal(voteCount, 1, "increments the candidate's vote count");
    })
  });
```

我们想在这里测试两件事：

1. 测试函数是否增加候选人的投票计数。
2. 测试投票者是否在投票时添加到映射中。

接下来，我们可以为函数的需求编写一些测试。让我们编写一个测试来确保我们的 vote 函数为双重投票抛出异常：

```
it("throws an exception for invalid candidates", function() {
    return Election.deployed().then(function(instance) {
      electionInstance = instance;
      return electionInstance.vote(99, { from: accounts[1] })
    }).then(assert.fail).catch(function(error) {
      assert(error.message.indexOf('revert') >= 0, "error message must
contain revert");
```

```
      return electionInstance.candidates(1);
    }).then(function(candidate1) {
      var voteCount = candidate1[2];
      assert.equal(voteCount, 1, "candidate 1 did not receive any votes");
      return electionInstance.candidates(2);
    }).then(function(candidate2) {
      var voteCount = candidate2[2];
      assert.equal(voteCount, 0, "candidate 2 did not receive any votes");
    });
  });
```

我们可以断言事务失败并返回错误消息。我们可以深入研究此错误消息，以确保错误消息包含“还原”子字符串。然后，我们可以通过确保候选人没有获得任何选票来确保我们的合同状态不变。

现在让我们编写一个测试来确保我们防止双重投票：

```
it("throws an exception for double voting", function() {
    return Election.deployed().then(function(instance) {
      electionInstance = instance;
      candidateId = 2;
      electionInstance.vote(candidateId, { from: accounts[1] });
      return electionInstance.candidates(candidateId);
    }).then(function(candidate) {
      var voteCount = candidate[2];
      assert.equal(voteCount, 1, "accepts first vote");
      // Try to vote again
      return electionInstance.vote(candidateId, { from: accounts[1] });
    }).then(assert.fail).catch(function(error) {
      assert(error.message.indexOf('revert') >= 0, "error message must
contain revert");
      return electionInstance.candidates(1);
    }).then(function(candidate1) {
      var voteCount = candidate1[2];
      assert.equal(voteCount, 1, "candidate 1 did not receive any votes");
      return electionInstance.candidates(2);
    }).then(function(candidate2) {
      var voteCount = candidate2[2];
      assert.equal(voteCount, 1, "candidate 2 did not receive any votes");
    });
  });
```

首先，我们将使用尚未投票的新帐户设置一个测试方案。然后我们将代表他们投票。然后我们将尝试再次投票。我们将断言此处发生了错误。我们可以检查错误消息，并确保没有候选人收到选票，就像之前的测试一样。

现在让我们运行我们的测试：

```
$ truffle test
Yay, they pass!
```

客户端投票

让我们添加一个表单，允许帐户在我们的“index.html”文件中的表格下方投票：

```
<form onSubmit="App.castVote(); return false;">
  <div class="form-group">
    <label for="candidatesSelect">Select Candidate</label>
    <select class="form-control" id="candidatesSelect">
    </select>
  </div>
  <button type="submit" class="btn btn-primary">Vote</button>
  <hr /></form>
```

让我们检查一下有关此表单的一些事项：

1. 我们使用空的选择元素创建表单。我们将在“app.js”文件中使用智能合约提供的候选人填充选择选项。
2. 表单有一个“onSubmit”处理程序，它将调用“castVote”函数。我们将在“app.js”文件中定义它。

现在让我们更新我们的 app.js 文件来处理这两件事。首先，我们在表单的选择元素中列出智能合约中的所有候选者。然后，我们将在帐户投票后隐藏页面上的表单。我们将更新渲染函数，如下所示：

```
render: function() {
  var electionInstance;
  var loader = $("#loader");
  var content = $("#content");

  loader.show();
  content.hide();

  // Load account data
  web3.eth.getCoinbase(function(err, account) {
    if (err === null) {
      App.account = account;
      $("#accountAddress").html("Your Account: " + account);
    }
  });

  // Load contract data
  App.contracts.Election.deployed().then(function(instance) {
    electionInstance = instance;
```

```
    return electionInstance.candidatesCount();
  }).then(function(candidatesCount) {
    var candidatesResults = $("#candidatesResults");
    candidatesResults.empty();

    var candidatesSelect = $('#candidatesSelect');
    candidatesSelect.empty();

    for (var i = 1; i <= candidatesCount; i++) {
      electionInstance.candidates(i).then(function(candidate) {
        var id = candidate[0];
        var name = candidate[1];
        var voteCount = candidate[2];

        // Render candidate Result
        var candidateTemplate = "<tr><th>" + id + "</th><td>" + name + "</td><td>" + voteCount + "</td></tr>"
        candidatesResults.append(candidateTemplate);

        // Render candidate ballot option
        var candidateOption = "<option value='" + id + "' >" + name + "</option>"
        candidatesSelect.append(candidateOption);
      });
    }
    return electionInstance.voters(App.account);
  }).then(function(hasVoted) {
    // Do not allow a user to vote
    if(hasVoted) {
      $('form').hide();
    }
    loader.hide();
    content.show();
  }).catch(function(error) {
    console.warn(error);
  });
}
```

接下来，我们要编写一个在提交表单时调用的函数：

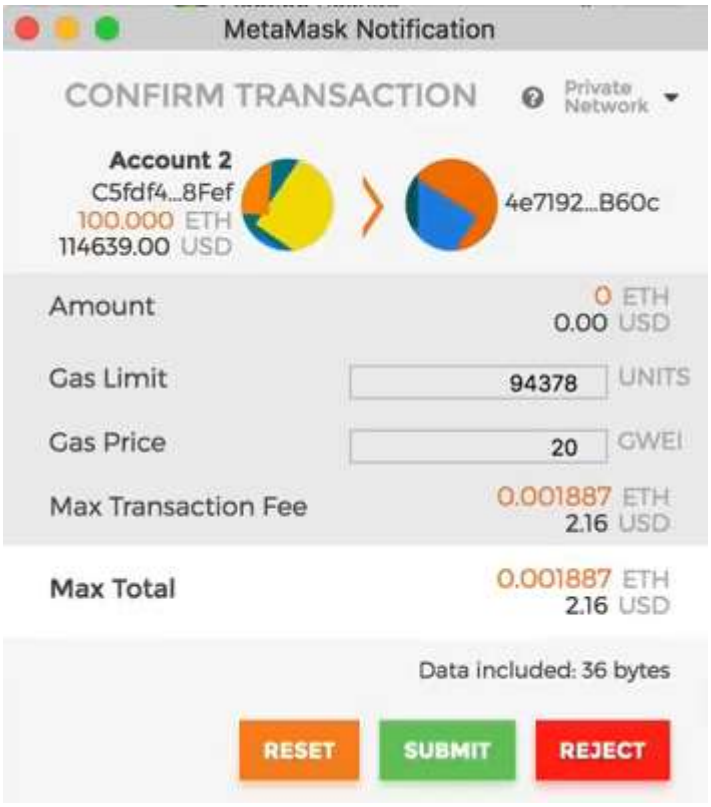
```
castVote: function() {
    var candidateId = $('#candidatesSelect').val();
    App.contracts.Election.deployed().then(function(instance) {
      return instance.vote(candidateId, { from: App.account });
    }).then(function(result) {
      // Wait for votes to update
```

```
      $("#content").hide();
      $("#loader").show();
    }).catch(function(err) {
      console.error(err);
    });
  }
```

首先，我们查询表单中的候选人 ID。当我们从智能合约调用投票函数时，我们传入此 ID，并为当前帐户提供该函数的“from”元数据。这将是一个异步调用。完成后，我们将显示加载器并隐藏页面内容。每当记录投票时，我们都会做相反的事情，再次向用户显示内容。

现在，前端应用程序应如下所示：

继续尝试投票功能。完成后，您应该会看到一个小狐狸钱包确认弹出，如下所示：



点击提交后，您已成功投票！您仍会看到加载屏幕。现在，您必须刷新页面才能看到记录的投票。我们将在下一节中实现自动更新加载程序的功能。如果遇到困难，可以在此处的教程中引用完整的客户端代码。

Watch Events - Step 4

本教程这一部分的随附视频片段从 1：48：05 开始。您可以在此处下载本教程此部分的代码。如果您遇到困难，请随时使用这些作为参考点！

本教程的最后一步是在投票时触发事件。这将允许我们在帐户投票时更新我们的客户端应用程序。幸运的是，这很容易。让我们首先在合约中声明一个事件，如下所示：

```
contract Election {
    // ...
    event votedEvent (
        uint indexed _candidateId
    );
    // ...
}
```

现在我们可以在“vote”函数中触发这个“voted”事件，如下所示：

```
function vote (uint _candidateId) public {
```

```
    // require that they haven't voted before
    require(!voters[msg.sender]);

    // require a valid candidate
    require(_candidateId > 0 && _candidateId <= candidatesCount);

    // record that voter has voted
    voters[msg.sender] = true;

    // update candidate vote Count
    candidates[_candidateId].voteCount ++;

    // trigger voted event
    votedEvent(_candidateId);
}
```

现在我们已经更新了合同，我们必须运行迁移：

```
$ truffle migrate --reset
```

我们还可以更新测试以检查此投票事件，如下所示：

```
it("allows a voter to cast a vote", function() {
  return Election.deployed().then(function(instance) {
    electionInstance = instance;
    candidateId = 1;
    return electionInstance.vote(candidateId, { from: accounts[0] });
  }).then(function(receipt) {
    assert.equal(receipt.logs.length, 1, "an event was triggered");
    assert.equal(receipt.logs[0].event, "votedEvent", "the event type is
correct");
    assert.equal(receipt.logs[0].args._candidateId.toNumber(),
candidateId, "the candidate id is correct");
    return electionInstance.voters(accounts[0]);
  }).then(function(voted) {
    assert(voted, "the voter was marked as voted");
    return electionInstance.candidates(candidateId);
  }).then(function(candidate) {
    var voteCount = candidate[2];
    assert.equal(voteCount, 1, "increments the candidate's vote count");
  })
});
```

此测试检查“vote”函数返回的交易收据，以确保它有日志。这些日志包含触发的事件。我们检查事件的类型是否正确，以及它是否具有正确的候选 ID。

现在，让我们更新客户端应用程序以侦听投票事件，并在触发该事件时触发页面刷新。我们可以使用这样的“listenForEvents”函数来做到这一点：

```
listenForEvents: function() {
  App.contracts.Election.deployed().then(function(instance) {
    instance.votedEvent({}, {
      fromBlock: 0,
      toBlock: 'latest'
    }).watch(function(error, event) {
      console.log("event triggered", event)
      // Reload when a new vote is recorded
      App.render();
    });
  });
}
```

此函数执行一些操作。首先，我们通过调用“votedEvent”函数来订阅投票事件。我们传入一些元数据，告诉我们监听区块链上的所有事件。然后我们“观看”这个事件。在这里，每当触发“votedEvent”时，我们都会登录到控制台。我们还会重新呈现页面上的所有内容。这将在记录投票后删除加载器，并在表格上显示更新的投票计数。

最后，我们可以在初始化合约时调用这个函数：

```
initContract: function() {
  $.getJSON("Election.json", function(election) {
    // Instantiate a new truffle contract from the artifact
    App.contracts.Election = TruffleContract(election);
    // Connect provider to interact with contract
    App.contracts.Election.setProvider(App.web3Provider);

    App.listenForEvents();

    return App.render();
  });
}
```

现在，您可以在客户端应用程序上投票，并实时观看记录的投票！请耐心等待，触发事件可能需要几秒钟。如果您没有看到活动，请尝试重新启动 Chrome。元狐狸钱包围绕事件存在已知问题。重新启动 Chrome 总是为我修复它。

祝贺！在此处下载本教程的完整源代码，并在此处观看完整视频。

对本教程满意吗？然后，**您需要**在此处加入我的免费培训，我将向您展示如何构建现实世界的区块链应用程序，以便您可以成为高薪区块链开发人员！

认识您的导师

--格雷